FTP_Algorithms

Cheat Sheet

Diego Gil

Herbstsemester 2024/25

Contents

1 Search and Analysis

2 Data Structures

2.1 Trees

2.1.1 KD-Trees

Problem Type: Construction of a KD-Tree from 2D points

What to Look For:

- Set of 2D points given as coordinates
- Request to build a KD-Tree
- Questions about tree properties (height, leaves)

Given Points: $P = \{(1,3), (12,1), (4,5), (5,4), (10,11), (8,2), (2,7)\}$

Solution Strategy:

1. Sort points by x-coordinate (root level)
2. Find median point
3. Split into left/right subtrees
4. Repeat with y-coordinates for next level
5. Continue alternating x/y until all points placed

Detailed Solution:

1. **Root Level (x-split)**
   - Sorted x:
     $(1,3), (2,7), (4,5),$
     $(\mathbf{5}, \mathbf{4}),$
     $(8,2), (10,11), (12,1)$
   - Median $(5,4)$ becomes root ℓ_1

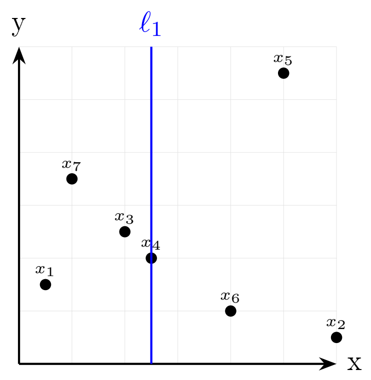


Figure 1: *
Coordinate Split at Root Level

2. **Tree Structure**

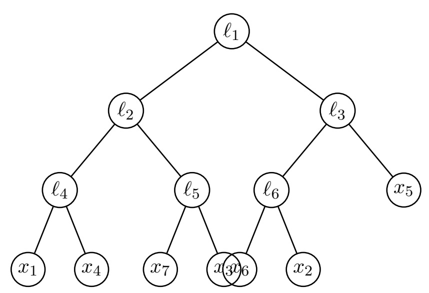


Figure 2: *
KD-Tree Structure

3. **Final Properties**
   - Height: 3 (counting from 0)
   - Leaves: 7 (all original points)
   - Second leaf from left: $(4,5)$

Exam Tips:

1. Always start by sorting points on current dimension
2. Mark median point clearly in your sorting
3. Draw coordinate system with splitting lines
4. Keep track of which dimension you're splitting on:
   - Level 0: x-coordinate
   - Level 1: y-coordinate
   - Level 2: x-coordinate
   - And so on...
5. Verify tree properties at the end

Common Mistakes to Avoid:

- Don't forget to alternate dimensions
- Don't skip sorting at each level
- Don't mix up left ($<$) and right ($>$) subtrees
- Don't forget to verify final tree properties

2.1.2 KD-Tree Complexity Analysis

Problem Type: Complexity proof for KD-Tree construction

What to Look For:

- Proof of time complexity $O(n \log n)$
- Proof of space complexity $O(n)$
- Recursive analysis

Solution Strategy:

1. Prove space complexity first (easier)
2. Analyze recursive structure
3. Set up recurrence relation
4. Apply Master Theorem

Space Complexity Proof:

1. For $n = 2^k$ points:
   - Internal nodes (parents): $2^k - 1$
   - Total nodes: $2^k + 2^{k-1} = n + n/2 = 3n/2 < 3n$
2. For general n (not power of 2):
   - Find t where $2^{t-1} < n < 2^t$
   - Internal nodes n_p: $2^{t-2} < n_p < 2^{t-1}$
   - Total nodes: $3 \cdot 2^{t-2} < n + n_p < 3 \cdot 2^{t-1}$
   - Therefore: $n + n_p < 3n$
3. Each node uses $O(1)$ storage
4. Total storage: $O(1) \cdot O(n) = O(n)$

Time Complexity Proof:

1. At each recursion:
   - Split n points into two subsets of $n/2$
   - Finding median costs $O(n)$

2. Recurrence relation:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

3. Apply Master Theorem:
   - Similar to Merge-Sort analysis
   - Results in $T(n) = O(n \log n)$

Key Points for Exam:

- Space complexity proof:
  - Count nodes for power of 2
  - Extend to general case
  - Multiply by constant storage
- Time complexity proof:
  - Identify recursive pattern
  - Write recurrence relation
  - Apply Master Theorem
- Remember median finding is $O(n)$

Common Mistakes to Avoid:

- Don't forget to account for non-power-of-2 cases
- Don't ignore constant factors in space analysis
- Remember to justify linear median finding
- Don't skip the Master Theorem application

3 Graph Algorithms

3.1 Graph Representations

3.1.1 Graph Transpose

Problem Type: Computing transpose G^T of a directed graph $G = (V, E)$

What to Look For:

- Graph representation type (matrix/list)
- Direction of edges must be reversed
- Time complexity analysis required

Key Definitions:

- $G^T = (V, E^T)$ where $E^T = \{(v, u) \mid (u, v) \in E\}$
- $|V| = n$ (number of vertices)
- $|E|$ (number of edges)

Solution for Adjacency Matrix:

1. Given matrix M_G, create M_G^T by swapping entries:

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{pmatrix}$$

$$M^T = \begin{pmatrix} m_{11} & m_{21} & \cdots & m_{n1} \\ m_{12} & m_{22} & \cdots & m_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1n} & m_{2n} & \cdots & m_{nn} \end{pmatrix}$$

2. Example:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, M^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

3. Time Complexity: $\Theta(n^2)$
   - Must swap $n^2 - n$ entries (excluding diagonal)
   - Each swap is $O(1)$

Solution for Adjacency List:

1. Create empty adjacency lists for G^T: $O(n)$
2. For each vertex v in G:
   - For each edge (v, w) in v's adjacency list
   - Add v to w's list in G^T
3. Time Complexity: $\Theta(|V| + |E|)$
   - Creating lists: $O(|V|)$
   - Processing edges: $O(|E|)$

Comparison:

- Matrix: $\Theta(n^2)$ always
- List: $\Theta(|V| + |E|)$ which is better for sparse graphs
- List requires more complex implementation

Common Mistakes to Avoid:

- Don't forget self-loops (diagonal elements)
- Don't count diagonal elements in matrix swaps
- Remember to initialize all new lists in adjacency list solution
- Don't confuse $|V|$ and $|E|$ in complexity analysis

3.2 Shortest Paths

3.2.1 Dijkstra's Algorithm Limitations

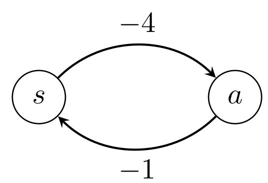
Problem Type: Counterexample for Dijkstra with negative weights

What to Look For:

- Directed graph with negative weights
- Minimal example showing algorithm failure
- Negative cycle demonstration

Solution:

1. Consider this directed graph:



2. Why Dijkstra fails:
   - Initial distance to a: -4
   - After one cycle: -5
   - After two cycles: -6
   - Continues to decrease indefinitely

Key Properties:

- Any negative cycle causes Dijkstra to fail
- Algorithm assumes:
  - Edge weights are non-negative
  - Shortest paths exist (no negative cycles)
- For negative weights, use Bellman-Ford instead

Common Mistakes to Avoid:

- Single negative edge isn't enough
- Example must have negative total cycle weight
- Remember: Bellman-Ford can detect negative cycles

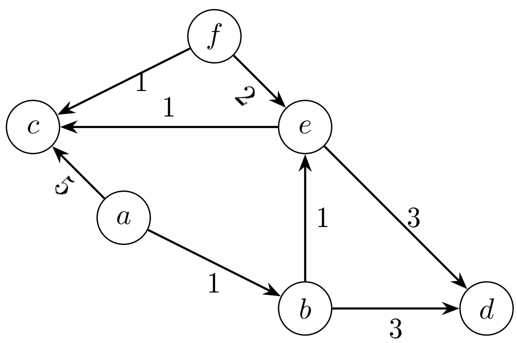
3.2.2 Dijkstra's Algorithm Step-by-Step

Problem Type: Tracing Dijkstra's algorithm iterations

What to Look For:

- Starting vertex (source)
- Number of iterations to analyze
- Distance values after specific iterations
- Edge weights and graph structure

Example Graph:



Initial State:

- Source vertex a: distance = 0
- All other vertices: distance = ∞
- No vertices marked as visited

After Two Iterations:

1. First Iteration:
   - Visit a
   - Update: $b.d = 1$, $c.d = 5$
2. Second Iteration:
   - Visit b (closest unvisited)
   - Update: $d.d = 4$, $e.d = 2$

Final State After 2 Iterations:

- Visited: $\{a, b\}$
- Distances:
  - $c.d = 5$ (via a)
  - $d.d = 4$ (via b)
  - $e.d = 2$ (via b)
  - $f.d = \infty$ (no path found yet)

Key Points:

- Always visit closest unvisited vertex
- Update distances through latest visited vertex
- Keep track of visited set
- Remember: distances are cumulative

Common Mistakes to Avoid:

- Don't forget to mark vertices as visited
- Only update distances through current vertex
- Check all edges from current vertex
- Remember to compare new paths with existing ones

3.3 BFS Properties

3.3.1 BFS Limitations with Shortest Paths

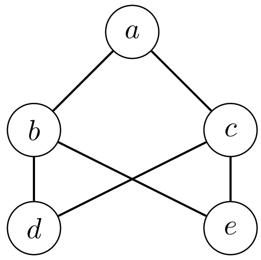
Problem Type: Counterexample showing BFS limitations

What to Look For:

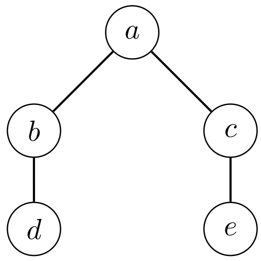
- Graph must have multiple shortest paths
- Some shortest paths cannot be discovered by BFS
- Example should be minimal

Solution:

1. Consider this undirected graph G:



2. Consider subgraph G' (a valid shortest path tree):



Key Observations:

- G' contains shortest paths from a to all vertices
- These paths are unique in G'
- BFS can never produce exactly G' because:
  - If it visits b before c: will use (b, e) instead of (c, e)
  - If it visits c before b: will use (c, d) instead of (b, d)
- No vertex ordering in BFS can produce G'

Important Properties:

- BFS always finds shortest paths
- But cannot find all possible shortest path trees
- Order of vertex processing affects which paths are found

- Some valid shortest path trees are impossible for BFS

Common Mistakes to Avoid:

- Don't confuse "shortest path" with "shortest path tree"
- Remember BFS guarantees shortest paths but not specific trees
- Example must work for all possible vertex orderings
- Graph should be minimal (removing edges breaks the property)